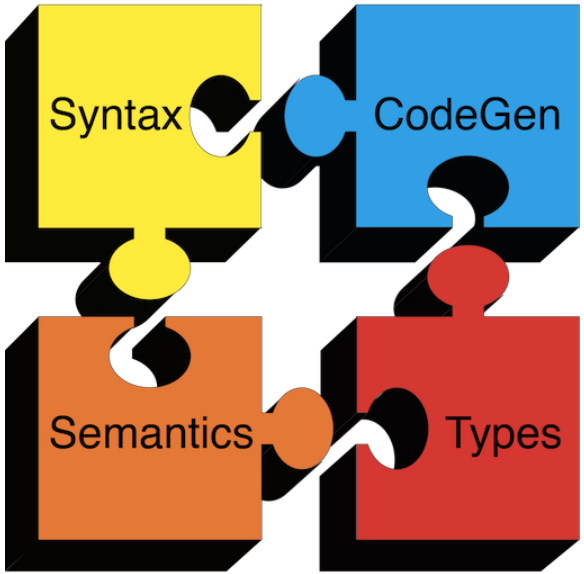

Compilers

Stack Machines

Stack Machines

- Only storage is a stack

- An instruction $r = F(a_1,\ldots a_n)$:
  - Pops n operands from the stack
  - Computes the operation F using the operands
  - Pushes the result r on the stack

Stack Machines

Stack Machines

- Consider two instructions
  - push i - push integer i on the stack
  - add - add two integers
  - A program:

```
push 7
push 5
add
```

Stack Machines

- Stack machines are a very simple machine model
  - Leads to a simple, small compiler
  - But not necessarily one that produces very fast code

Stack Machines

- Location of the operands/result is not explicitly stated
  - Always the top of the stack

- In contrast to a *register machine*
  - add instead of add r_1, r_2, r_3
  - More compact programs

- One reason that Java bytecode uses stack evaluation

Stack Machines

- There is an intermediate point between a pure stack machine and a pure register machine
- An *n-register stack machine*
  - Conceptually, keep the top *n* locations of the pure stack machine's stack in registers
- Consider a 1-register stack machine
  - The register is called the *accumulator*

Stack Machines

- In a pure stack machine
  - An add does 3 memory operations
  - Two reads and one write to the stack

- In a 1-register stack machine the add does

$$acc \leftarrow acc + top_of_stack$$

Stack Machines

- Consider an expression $op(e_1,\ldots,e_n)$
  - Note $e_1,\ldots,e_n$ are subexpressions
- For each e_i $(0 < i < n)$
  - Compute e_i
  - Push result on the stack
- Pop n-1 values from the stack, compute op
- Store result in the accumulator

Stack Machines

Stack Machines

After evaluating an expression e, the accumulator holds the value of e and the stack is unchanged.

Expression evaluation preserves the stack.

Stack Machines

Code	Acc	Stack
acc ← 3	3	<init>
push acc	3	3, <init>
acc ← 7	7	3, <init>
push acc	7	7, 3, <init>
acc ← 5	5	7, 3, <init>
acc ← acc + top_of_stack	12	7, 3, <init>
pop	12	3, <init>
acc ← acc + top_of_stack	15	3, <init>
pop	15	<init>

Stack Machines

Given the current state of the stack and accumulator, what is the next line of code to generate for the code fragment (2 * 3) + 5?

Current:

Acc : 5

Stack: 6,<init>

- push acc
- pop
- acc ← 6
- acc ← acc + top_of_stack